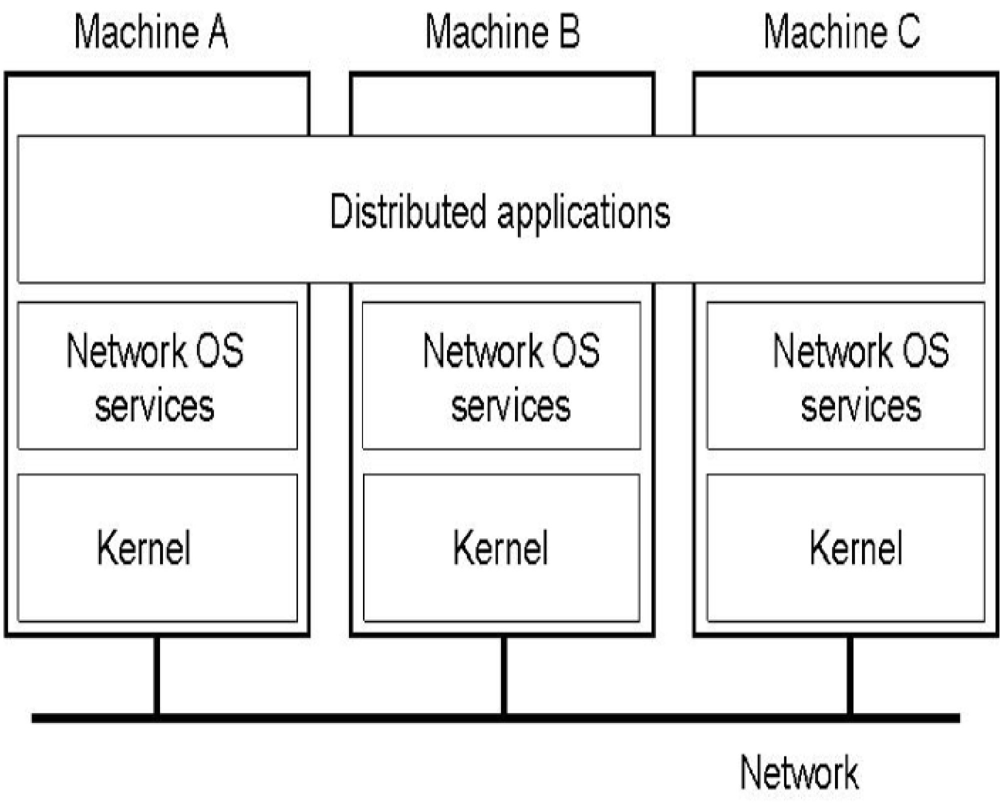
Modelling

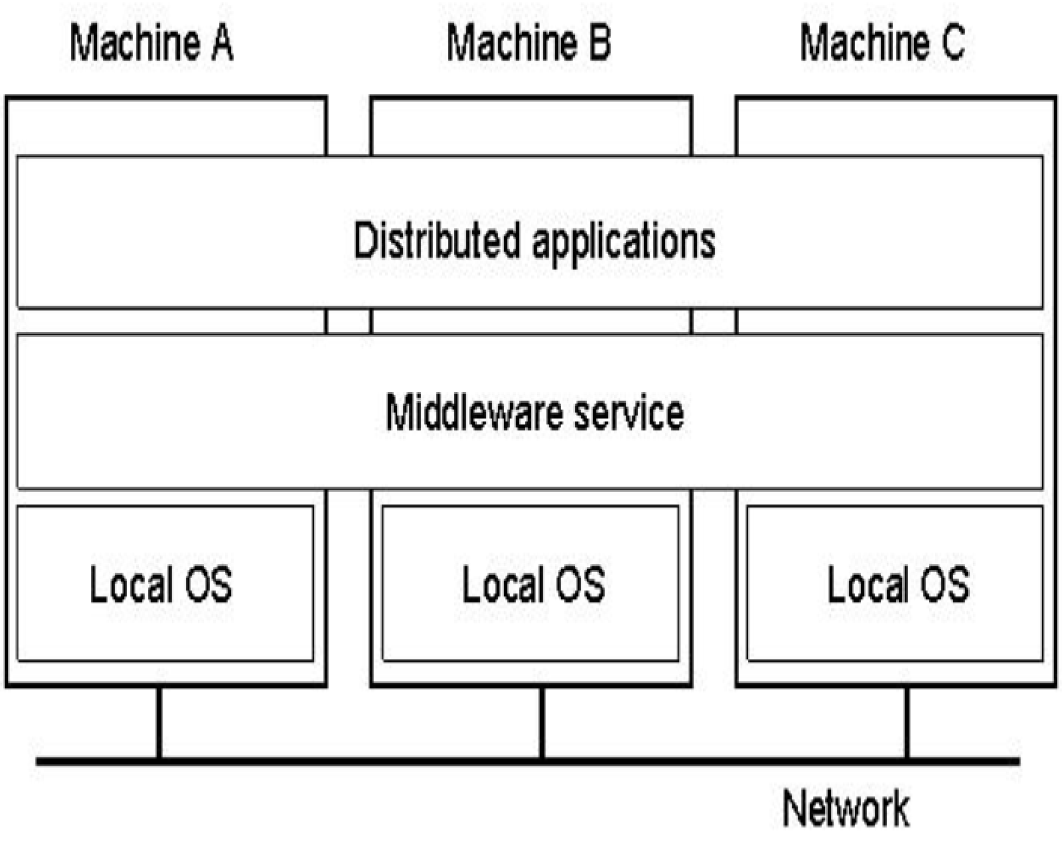
The software architecture of a distributed system

Distributed systems use two main kinds of software architectures to accomplish their goal:

- **Network OS**-based: the network operating system provides the communication services. Different machines may have different network OSs and masking platform differences is up to the application programmer



- **Middleware**-based: a middleware provides advanced communication, coordination and administration services, masking most of the platform differences



Middleware: a functional view

Middleware provides "*business-unaware*" services through a standard API, which raises the level of the communication activities of applications.

A middleware usually provides:

- **Communication** and **coordination** services, which can be:
  - Synchronous or asynchronous
  - Point-to-point or multicast
  - Masking differences in the network OS
- **Special application** services, like distributed transaction management, groupware and workflow services, messaging services, notification services and mode
- **Management** services, like naming, security, failure handling and more

---

The run-time architecture of a distributed system

The run-time architecture of a distributed system identifies the classes of components that build the system, the various types of connectors and the data types exchanged at run-time. Modern distributed systems often adopt one among a small set of well-known architectural styles:

- Client-server
- Service oriented
- REST
- Peer-to-peer[1]
- Object-oriented
- Data-centered
- Event-based
- Mobile code

The client-server architecture

The most common architectural style today is the **client-server** architecture, in which different components in the system have different roles: the servers provide a set of services through a well defined API, while the users access those services through clients. Communication is generally message based, but it can also be based on **Remote Procedure Call** (*RPC*).

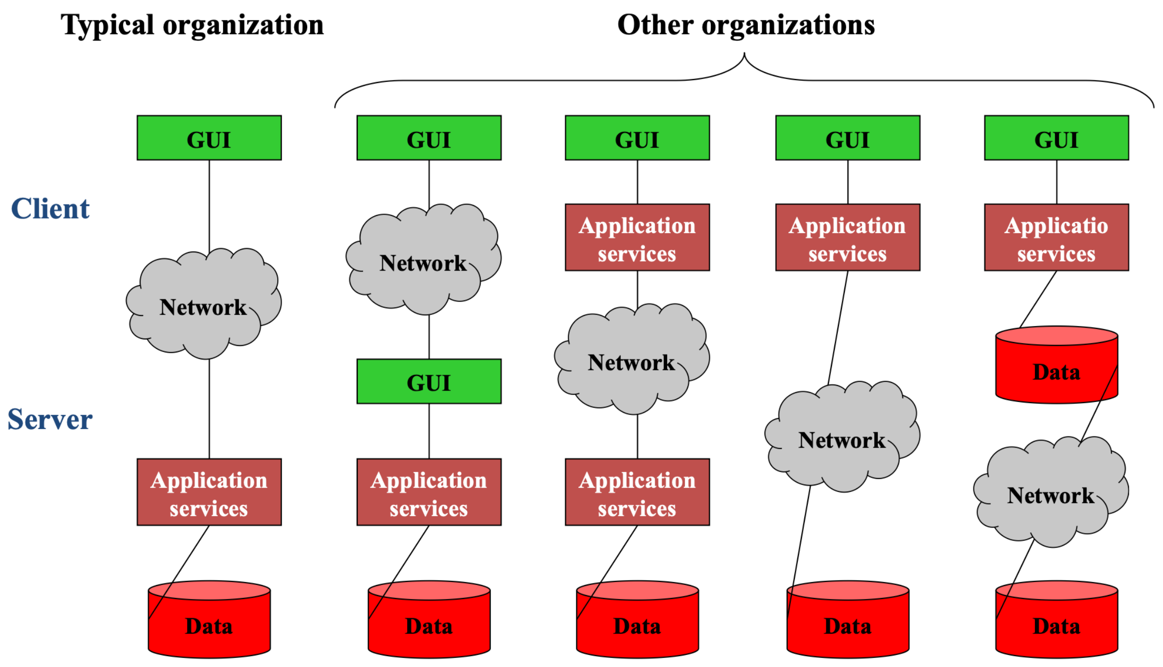
Often, servers operate by taking advantage of the services offered by other distributed components. In that case, we have a **three-tiered client-server architecture**.

The services offered by a distributed application can be partitioned in three classes:

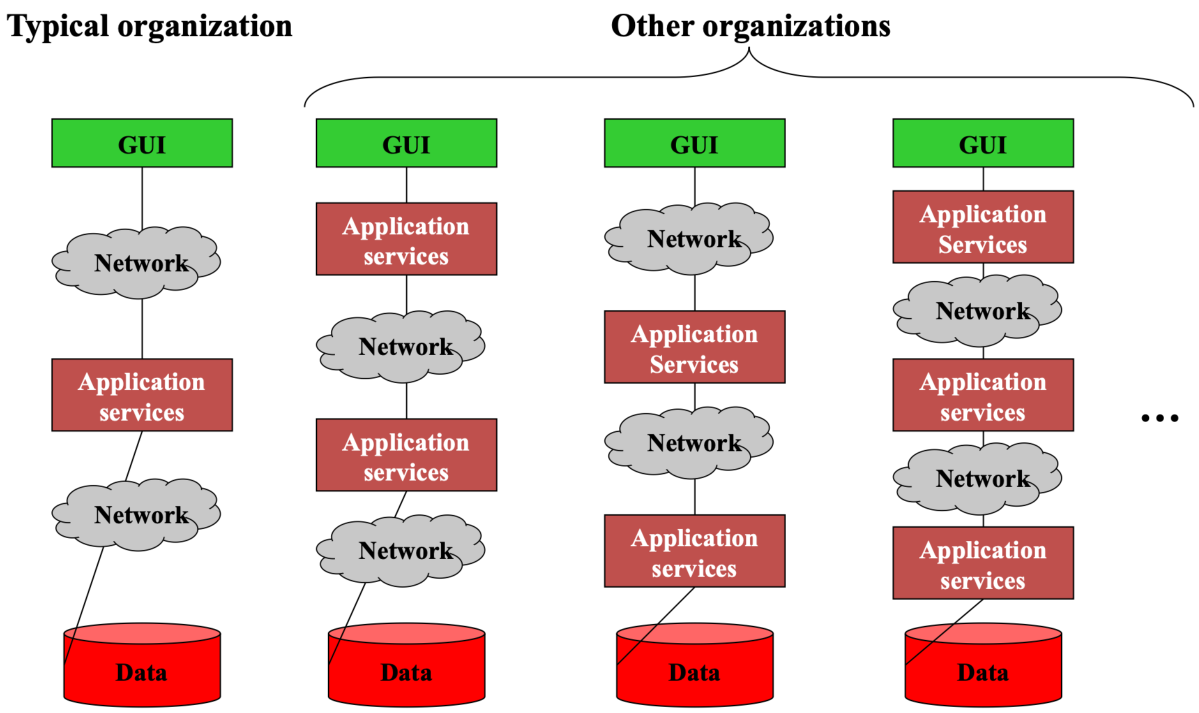
- User interface services
- Application services
- Storage services

Multi-tiered client-server applications can be classified looking at the way such services are assigned to different tiers.

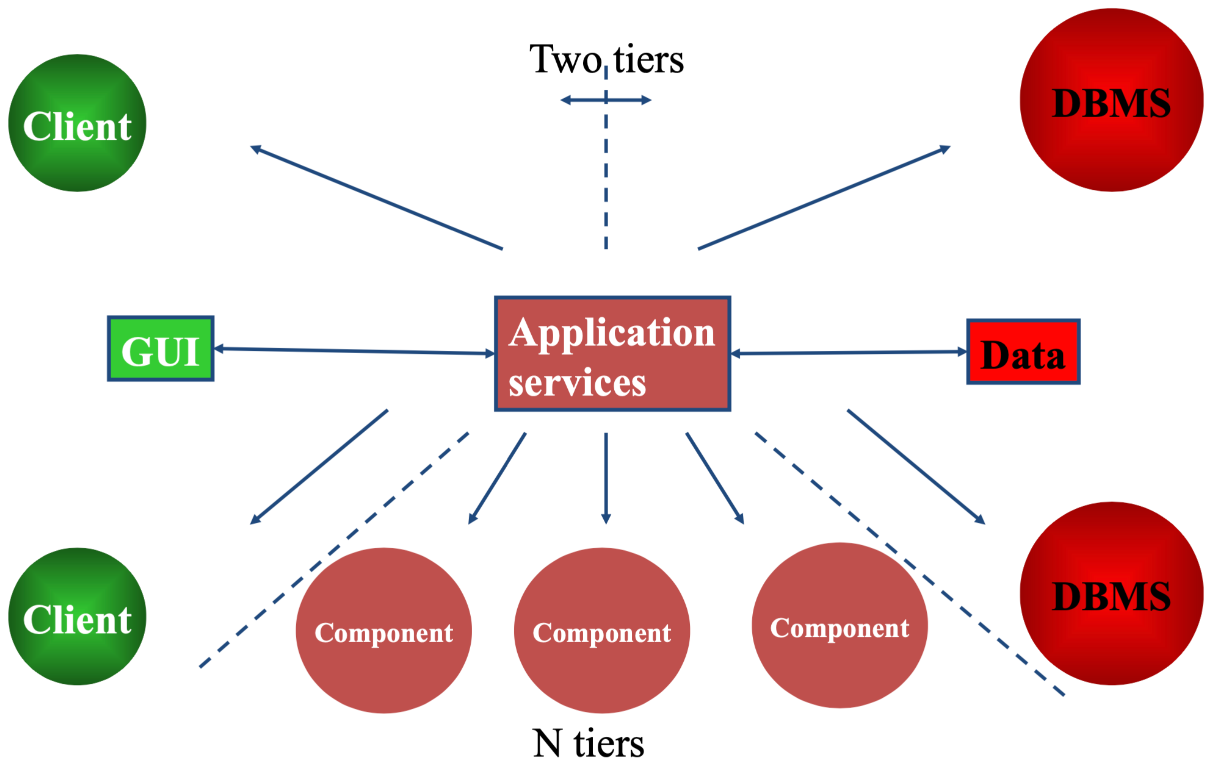
The standard two-tiered architectures are found in the image below:



While the standard three-tiered architectures are described in the following picture:

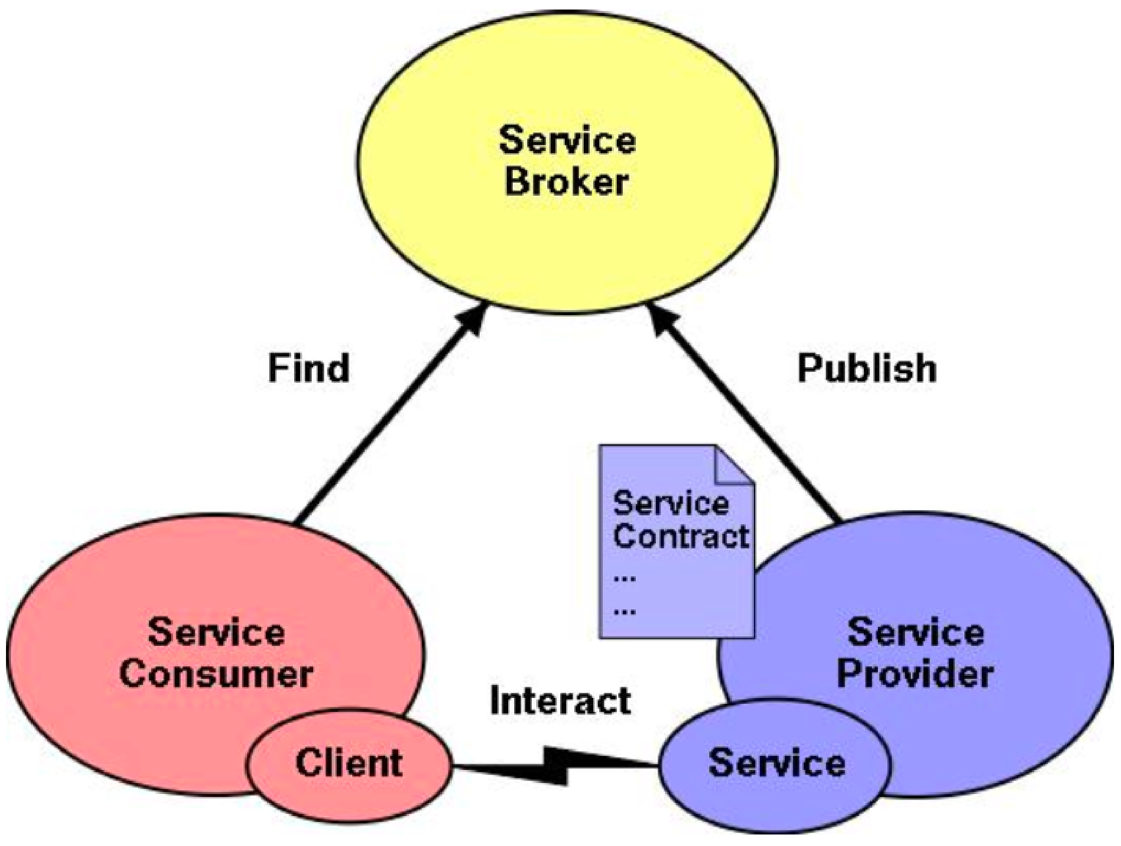


Recently, n-tiered architectures have appeared. They are extensions of the ones above:



The service-oriented architecture

Service-oriented architectures are built around the concepts of services, service providers, service consumers and service brokers:



Services represent loosely coupled units of functionality which are exported by **service providers** to **service brokers**. Brokers hold the description of available services to be **searched by interested consumers**, which bind and invoke the services they need.

Orchestration is the process of invoking a set of services in an ad-hoc workflow to satisfy a given goal.

There are several incarnations of service-oriented architectures nowadays, such as:

- OSGI
- JXTA
- Jini
- Web Services

Web services

According to the W3C:

> A web service is a software system designed to support interoperable machine-to-machine interaction over a network.

Web service interfaces are described via **WSDL** (*Web Service Description Language*), which includes a set of operations exported by the web service.

Web service operations are invoked through **SOAP**, a protocol based on XML which defines the way messages are exchanged between consumers and providers. SOAP is usually based on HTTP, but other transport protocols can also be used.

UDDI (*Universal Description Discovery and Integration*) describes the rules that allow web services to be exported and searched through a registry.

The REST style

REpresentational State Transfer (*REST* in short) is both a nice way to describe the web and a set of principles that define how web standards are supposed to be used.

Key goals of REST include:

- Scalability of component interactions
- Generality of interfaces

- Independent deployment of components
- Intermediary components to reduce latency, enforce security and encapsulate legacy systems

REST has some constraints however:

- Interactions are client-server only
- Interactions are stateless, so state must be transferred every time
- The data within a response to a request must be implicitly or explicitly labeled as cacheable or non-cacheable
- Each component cannot see beyond the immediate layer which they are interacting with
- Clients must support code-on-demand
- Components expose a uniform interface

The uniform interface exposed by components must satisfy four constraints:

1. **Identification of resources**: each resource must have an ID (usually a URI) and everything that has an ID must be a valid resource.
2. **Manipulation of resources through representations**: REST components communicate by transferring a representation of a resource in a format that matches one of an evolving set of standard data types selected dynamically based on the capabilities or desires of the recipient and the nature of the resource. Whether the representation is in the same format as the raw resource or is derived from the resource remains hidden behind the interface. A representation consists, in fact, of data and metadata describing the data.
3. **Self-descriptive messages**: control data defines the purpose of a message between components, such as the action being requested or the meaning of a response. It is also used to parametrise requests and override the default behaviour of some connecting elements.
4. **Hypermedia as the engine of an application state**: clients move from a state to another each time they process a new representation, usually linked to other representations through hypermedia links.

Peer-to-peer

In a peer-to-peer[1-1] application, all components play the same role: there is no distinction between clients and servers.

P2P was born because the client-server architecture does not scale well and a server is also a single point of failure for a system. P2P leverages the increased availability of broadband connectivity and processing power at the end-host to overcome such limitations.

P2P promotes the sharing of resources and services through a direct exchange between peers. Resources can be of many kinds:

- Processing cycles
- Collaborative work
- Storage space
- Network bandwidth
- Data

P2P is different because it "takes advantage of the resources at the edges of the network".

Object-oriented

In object-oriented architectures, the distributed components encapsulate a data structure providing an API to access and modify it. Each component is responsible for ensuring the integrity of the data structure it encapsulates, while the internal organisation of such data structure is hidden to the other components.

Components in object-oriented architectures interact through RPC.

Object-oriented architectures are based on the P2P model, but are often used to implement client-server applications.

The object-oriented architecture has a number of pros going for it:

- Information hiding hides complexity in accessing and managing shared data
- Encapsulation and information hiding reduce the management complexity
- Objects are easy to reuse among different applications
- Legacy components can be wrapped within objects and easily integrated in new applications

Data-centered

In data-centred architectures, components communicate through a common, usually passive, repository, from which data can be added or removed.

Communication with the repository is usually handled with RPC and access to the repository is typically synchronised.

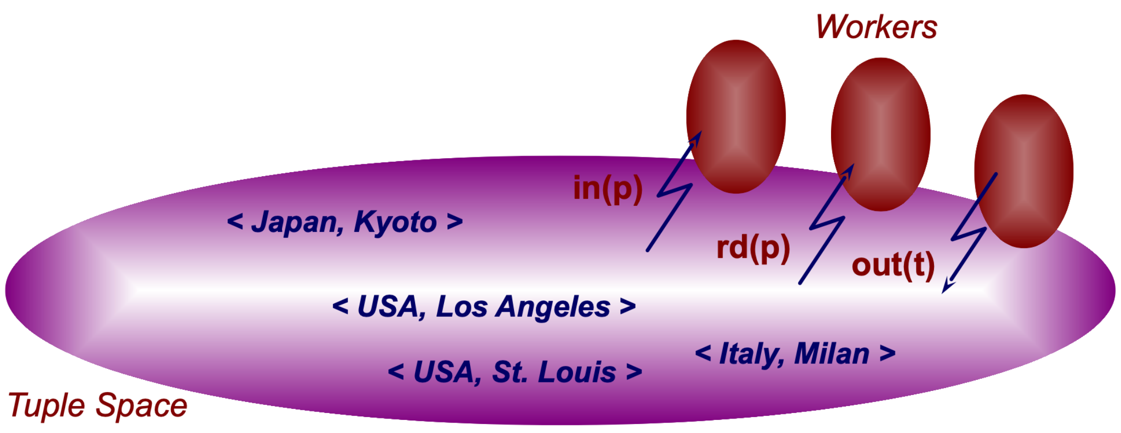
Linda and tuple spaces

Linda is a data sharing model proposed in the 80s by Carriero and Gelernter and it's mostly used for parallel computation. It has recently been revitalised in the context of distributed computing with:

- IBM TSpaces
- Sun JavaSpaces
- GigaSpaces

In Linda, communication is persistent, implicit, content-based and generative.

Linda is an architecture with a very high degree of decoupling.



In Linda, data is contained in ordered sequences of typed fields, called **tuples**. These tuples are stored in a persistent, global, shared space, called **tuple space**.

The standard operations available in Linda's tuple space are:

Operation	Description
`out(t)`	Writes the tuple `t` in the tuple space
`rd(p)`	Returns a copy of a tuple matching the pattern (or template) `p`
`in(p)`	Similar to `rd(p)`, but withdraws the matching tuple from the tuple space
`eval(a)`	Inserts the tuple generated by the execution of the process `a`

Many variants of these primitives are also available, depending on implementation. Some of the non-standard primitives have non-trivial distributed system implementations: e.g. if atomicity has to be preserved, probes require a distributed transaction.

The tuple space model is not easily scaled on a wide-area network, since it is very difficult to store and replicate tuples efficiently and to route queries well.

Linda's model is also only proactive: processes explicitly request a tuple query, so reactive and/or asynchronous behaviour must be implemented with an extra process and blocking operations.

For the above reasons, commercial implementations of Linda only provide client access to a server holding the tuple space instead of a fully distributed, decentralised implementation.

Event-based

In event-based architectures, components collaborate by exchanging information about occurring events. In particular, components **publish** notifications about the events they observe or they **subscribe** to the events they are interested about.

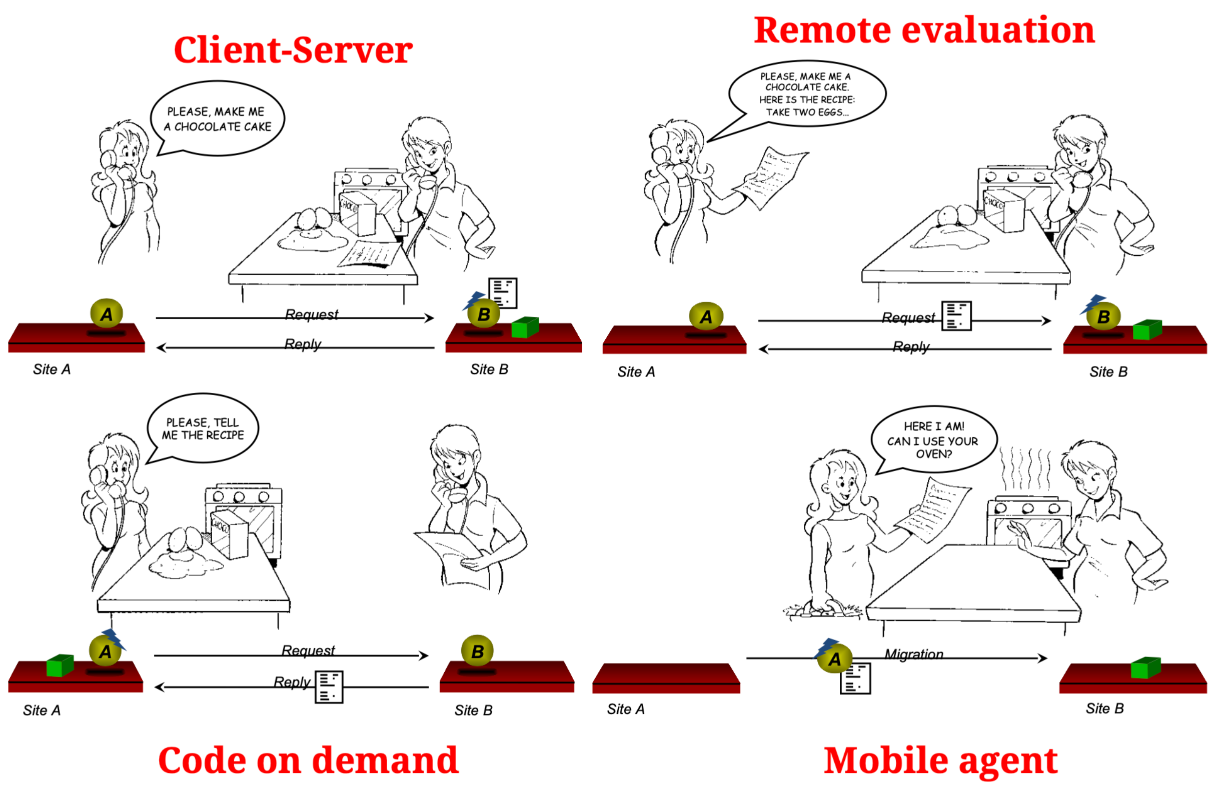
In event-based architectures, communication is:

- Purely message based
- Asynchronous
- Multicast
- Implicit
- Anonymous

Mobile code

Mobile code architectures are based on the ability to relocate the components of a distributed application **at run-time**. There are different implementations of this architecture, depending on the original and final location of resources, know-how and computational components.

The main mobile code paradigms are exemplified by the picture below:


Client-Server
PLEASE, MAKE ME A CHOCOLATE CAKE
A
Request
Reply
B
Site A
Site B
Remote evaluation
PLEASE, MAKE ME A CHOCOLATE CAKE. HERE IS THE RECIPE: TAKE TWO EGGS...
A
Request
Reply
B
Site A
Site B
PLEASE, TELL ME THE RECIPE
A
Request
Reply
B
Site A
Site B
Code on demand
HERE I AM! CAN I USE YOUR OVEN?
A
Migration
Site A
Site B
Mobile agent

In mobile code architectures, we distinguish between **strong** and **weak mobility**:

Mobility	Description
Strong	The ability of a system to allow the migration of both the code and the execution state of an executing unit to a different computational environment. Very few systems provide this.
Weak	The ability of a system to allow code movement across different computational environments. This is provided by several mainstream systems, including Java, .NET and Web.

In practice, mobile code has the advantage of being able to move pieces of code or even entire components at runtime, which provides great flexibility for programmers. However, securing mobile code application is extremely difficult.

The interaction model

Distributed algorithms

Traditional programs can be described in terms of the algorithm they implement. In this case, steps are strictly sequential and execution speed only influences performance (typically).

Distributed systems, however, are composed of many processes, which interact in complex ways. The behaviour of a distributed system is described by a **distributed algorithm**, which is a definition of the steps taken by each process, including the transmission of messages between them.

Furthermore, the behaviours of a distributed system are influenced by several factors, such as:

- The rate at which each process executes
- The performance of the communication channels
- The different clock drift rates

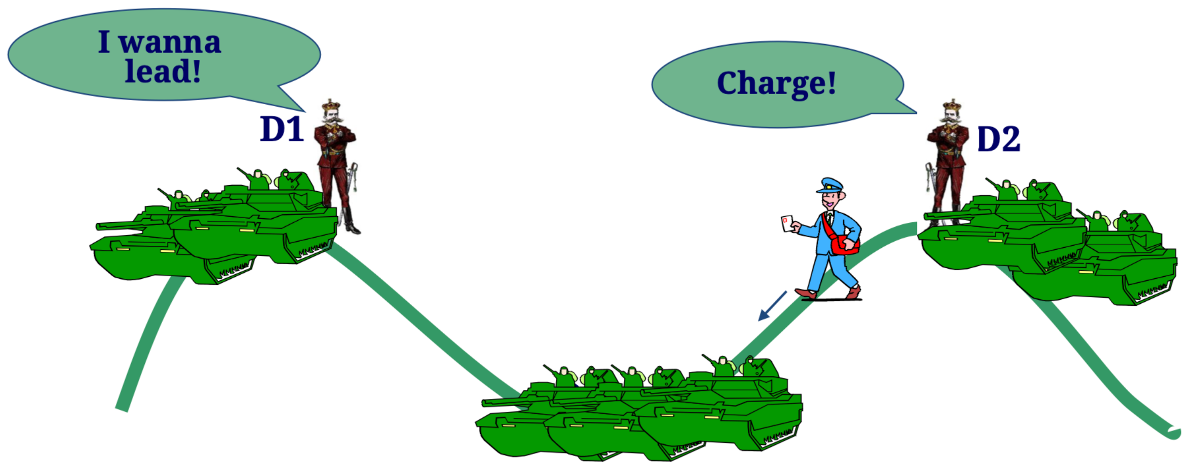
To formally analyse the behaviour of a distributed system, we must distinguish between **synchronous** and **asynchronous** distributed systems:

Type of system	**Description**
Synchronous	The time to execute each step of a process has known lower and upper bounds; each message transmitted over a channel is received within a known bounded time; each process has a local clock whose drift rate from real time has a known bound.
Asynchronous	There are no bounds for process execution speeds, message transmission delays or clock drift rates.

Any solution that is valid for an asynchronous distributed system is also valid for a synchronous one, but the viceversa is false.

The pepperland example

Let's hypothesise about two military divisions: the pepperland divisions:



Both divisions are safe as long as they remain in their encampments. If both charge the enemy at the same time, whey win; otherwise, they lose.

In order to win, the generals of each division have to agree on:

- Who will lead the charge
- When the charge will take place

For this example, we suppose that messengers can travel from one division to the other without being captured by the enemy.

In *asynchronous* pepperland, as well as in *synchronous* pepperland, it is possible to agree on who will lead the charge. However, charging together is a different issue: it is **simply impossible in asynchronous pepperland**. In fact, if the leader sends a messenger to the other general telling them to charge, the messenger may take an undefined amount of time to reach him. Moreover, differences on each division's clock do not allow for strategies based on sending a message with the time in which the charge will occur.

In *synchronous* pepperland, though, it is possible to determine the maximum difference between charge times. If min and max are respectively the minimum and maximum transmission times, the leader sends the "charge" message to the other general, waits for min minutes and then charges; as soon as the other general receives the "charge" message, he charges with his own division.

While the second division may charge later than the first one, the maximum time difference is known: it's $\Delta t = \text{max} - \text{min}$. If we know that the charge will last longer than this time difference, victory is guaranteed.

The failure model

In distributed systems, both processes and communication channels may fail. The failure model in a distributed system defines the ways in which failure may occur to provide a better understanding of the effects of failures.

We will distinguish between three types of failures:

Type of failure	Description
Omission	Processes fail stop or crash, while channels do not send or receive messages

Type of failure	Description
Byzantine (or arbitrary)	Processes may omit intended processing steps or add more, while channels may deliver corrupted messages, non-existent messages or even deliver the same message more than once
Timing	This only applies to synchronous systems and occurs when one of the time limits defined for the system is violated

Failure detection in pepperland

Let's return to the pepperland example for a moment. How could we detect if one of the two divisions has been attacked and defeated by the enemies?

In synchronous pepperland, it is trivial: each division can send a messenger to the other one saying they're still undefeated. If no messengers arrive for longer than a predefined amount of time, one division assumes the other one has been defeated.

In asynchronous pepperland however, we cannot distinguish weather the other division has been defeated or the message is just taking a very long time to arrive to its destination.

Impossibility of distributed consensus in practice

The impossibility of distributed consensus was formally demonstrated by Fischer, Lynch and Patterson in 1985 and is usually referred to as the **FLP Theorem**.

This theorem has real life implications, for example:

- Committing or aborting a transaction in a distributed database
- Agreeing on values of replicated, distributed sensors
- Agreeing on whether a system component is faulty

In practice, this is solved by either changing the assumptions (e.g. rendering links reliable enough) or by reducing guarantees (e.g. only making probabilistic guarantees instead of deterministic ones).

---

1. Also look at 8. Peer-to-peer. ↩ ↩